CMSC 417 Programming Assignment #3

Due October 24, 2001 (10:00 AM)

Introduction

In this project, you will implement a link state routing system. Each node in the network will be connected to one or more other nodes. Using hello packets and bounded flooding, you will implement a system to inform all of the nodes in the network of the current network topology. Each node will also build a routing table that minimizes the number of hops to all other nodes in the network.

This project will be the first introduction to several key ideas that will later in the semester. It will also combine what you have learned (and probably most of the code) from projects one and two.

Timers

A key building block for this project will be the timer thread. By using the timer thread, you will be able to cause events to happen in your system at specific times in the future (for example sending hello packets at a fixed interval of time).

The implementation for the timer thread will be via the queue abstraction you built for project two. When a thread wishes to schedule an event in the future, it enqueue's a request to the timer thread. When the requested time interval has elapsed, the timer thread enqueues a message indicating that the time interval has finished.

For the timer thread, you will need to implement the following interface:

`int createAlarm(int expireTime, Queue *respQueue, bool repeats)`

> Create a new alarm that will enqueue a response into the queue respQueue expireTime milliseconds in the future. If the repeats parameter is true, it will en-queue a response **every** expireTime milliseconds until canceled. If the return value is positive, it contains the unique identifier of the alarm event, and if it is negative it indicates an error in creating the alarm.

`void cancelAlarm(int alarmId)`

> Cancel the passed alarm. It is possible that one or more alarm notifications may occur after the alarm is canceled due to alaram events passing the cancelAlarm request in different queues.

Note: Alarm ids should be globally unique within a single router and will likely be allocated by the thread making the request for an alarm. Thus, don't forget to include appropriate mutex operations in your createAlarm routine.

Messages

In this project you will need to exchange messages between different threads in your system. In order to allow a single thread to respond to a variety of different message types such as a timer event or the arrival of a data packet, you will need to define a common message format to pass via the queues. Each message should start with an integer field that indicates the type of the message (e.g. TIMER_REQ, TIMER_ACK, TIMER_EXPIRE, etc.). In addition, each message can contain a variety of parameters that are specific to that message (e.g. when the alarm should expire). You should define a C struct that includes a field for the message type, and a union for any message specific fields that are required.

Nodes

Each "node" in your network is a UNIX process that will receive data a specific port from its neighbors, and send data to its neighbors on their designated ports. When a node starts running, it calls the routine `config_get_info`. This routine (we will supply to you in the files config-net.h and config-net.c) returns the hosts list of neighbors (expressed as hostname, port pairs).

While the connectivity of the network may change over time due to node failures, the topology of the underlying "physical network" as specified by the set of links joining the virtual nodes remains constant. This topology is specified in a network configuration file given to your program on the command line. This file specifies the virtual nodes in the network (i.e., the machine / UDP port pairs that run your program) and the links between them. The format of the network configuration file is shown below. You should use ports from your assigned ports for each node. The numbers after the links statement are the node numbers that are directly connected to that node.

```
Node fe:90::01 (tracy, 5000) links fe:90::02

Node fe:90::02 (tracy, 5001) links fe:90::03 fe:90::04

Node fe:90::03 (tracy, 5002) links fe:90::02 fe:90::04

Node fe:90::04 (tracy, 5003) links fe:90::02 fe:90::03
```

The files config-net.{c,h} also define to helper routines:

`int parseV6Addr(char *str, v6addrType v6addr)`

This function parses a string ipv6 address into an array of 16 octets. It returns 1 on success and 0 on failure. It supports the :: shorthand syntax f810::0001.

`void unparseV6Addr(char *str, v6addrType v6addr)`

This function converts an IPv6 address (passes as an array of 16 octets) back into a string.

Building Routing Tables

Every HELLO_INTERVAL mili-seconds, a node sends a hello packet to all of its neighbors. When a host receives a HELLO packet, it records the link as operational. If no HELLO packet has been received from a neighboring host in the past DEAD_INTERVAL mili-seconds, it assumes the link (or host) has failed. Every TOPOLOGY_INTERVAL mili-seconds, a host sends it current list of active neighbors to all other nodes in the network (via a flooding message). Every ROUTE_UPDATE_INTERVAL mili-seconds, a host re-computes its routing tables based on its current knowledge of the network topology.

The routing tables should be computed using Dikstra's algorithm. When a new routing table is built, it should be printed to the screen. Also, when new topology information (not currently available to that node) arrives it should also be printed. Each node will have a routing table entry for every other node in the network, and information about the first hop to that node (i.e. the node number of the first hop).

Parameters File

The parameters for network operation are read from a configuration file at the start of the node operation. For this project the required parameters (and their default values) are:

Variable	**Default Value**	**Description**
HELLO_INTERVAL	500	Frequency of Hello packets
DEAD_INTERVAL	2500	Interval to consider a node down
TOPOLOGY_INTERVAL	1000	Topology flooding frequency
ROUTE_UPDATE_INTERVAL	10000	Frequency of re-building routing table

The parameters in the file are represented one per line. Each line is a variable name, a space, and then an integer value for the parameter value. A line that starts with # is a comment line, and it should be ignored.

Threads

This project will have a variety of threads that perform specific functions. I would suggest having three threads (in addition to the initial thread). One thread should manager timer events, the second thread should receive packets from the network and en-queue them to the appropriate handler thread. The third thread would be responsible for building routing tables. It might make sense to split the routing thread into two separate threads, one for maintaining topology information, and the other for building the routing tables.

Packet Format

Each message you send over the wire (between hosts) should be an IPv6 packet. The format of an IPv6 packet is:

Field	Size (in bytes)	Notes
Version	0.5	Always 6
Priority	0.5	
Flow Label	3	Always zero
Payload Length	2	In bytes
Next Header	1	See table
Hop Limit	1	
Source Address	16	
Destination Address	16	

IPv6 Header Format

In addition, you will need to generate packets in specific formats. During the rest of the project, you will need to be able to generate ICMP, POSPF, and PTCP packets. In particular for this project, you will need to generate POSPF (Pseudo Open Shortest Path) packets, the routing protocol for CMSC417. You will design the specific format for these packets to meet your needs. The protocol type numbers, however, must conform to the types in this table:

Value	Protocol	Description
58	ICMP	Control (ping)
101	POSPF	Routing (for 417)
106	PTCP	Transport protocol

Extension (Next) Header Values

Implementation Requirements

You should submit a tar file that contains the source code for your implementation of the queue abstraction and routing code. Like the earlier programs, you should submit a tar file. The tar file should include a Makefile that compiles your code.

You should also submit a script file for **each** of the nodes using the configuration files supplied.

Each node should read its topology configuration file from the file name `network.config` in the current directory where the node it started (the routine config_get_info handles this). It should also read the parameters file `network.parameters` from the current working directory.